Figure 1

```
module EXAMPLE:

input RESET, START; output GOT;

signal REQUEST, GRANT in
  loop abort                        % RESET restarts the loop
    await START;
    emit REQUEST;
    present GRANT then emit GOT end
  ||                                % run concurrently
    loop
      present REQUEST then emit GRANT end;
      pause;                        % wait for the next cycle
      pause
    end
  when RESET end
end.
```

Figure 2A

p ; q	p; q;

Figure 2B

emit S	S = 1;

Figure 2C

loop p end	for (;;) p;

Figure 2D

present S then p else q end	if (S) p; else q;

Figure 3A

```
pause                       state = k;
                            if (level < 1) level = 1;
                            goto Join;
                        case k:
```

Figure 3B

```
await S                     goto Entry;
                        case k:
                            if (!S) {
                            Entry:
                                    state = k;
                                    if (level < 1) level = 1;
                                    goto Join;
                            }
```

Figure 3C

```
                            goto Entry;
                        case k:
abort                       if (!S)
                                    switch (state) {
        body                                Entry: body;
when S                              }
```

Figure 3D

```
                            goto Entry;
                        case k:
suspend                     if (S) {
                                    if (level < 1) level = 1;
                                    goto Join;
                            }
                            switch (state) {
        body                        Entry: body;
when S                      }
```

Figure 3E

```
signal S in                     S = 0;
                                goto Entry;
                            case k:
                                S = 0;
                                switch (state) {
        body                        Entry: body;
end                             }
```

Figure 3F

```
exit T;                     if (level < 2) level = 2;
                            goto Join;
```

Figure 3G

```
trap T in
        bodyA
||
        bodyA
handle T do
        handler
end
```

```
        innerLevel = 0;
        fork StartA, StartB;
case k:
        innerLevel = 0;
        fork ResumeA, ResumeB;

ResumeA:
        switch (statep) {
        StartA: bodyA;
        case 0: ;
        }
        goto InnerJoin;

ResumeB:
        switch (stateq) {
        StartB: bodyB;
        case 0: ;
        }
        goto InnerJoin;

InnerJoin:
        join;
        switch (innerLevel) {
        case 1:         /* paused */
                state = k;
                if (level < 1) level = 1;
                goto OuterJoin;
        case 2:         /* exited */
                handler;
                break;
        }
```

Figure 4

```
                              Start: goto L0;
                              Resume:
                              switch (s & 0x3) {
pause;                        L0:    s=1; goto Join;
pause;                        case 1: s=2; goto Join;
                              case 2: goto L1;
abort                         case 3: if (!B)
                                  switch (s>>2 & 0x7 ) {
    pause;                        L1:    s=3 | 0<<2; goto Join;
    pause;                        case 0: s=3 | 1<<2; goto Join;
                                  case 1: goto L2;
    abort                         case 2: if (!A)
                                      switch (s>>5) {
        pause;                        L2:    s=3 | 2<<2 | 0<<5; goto Join;
        pause                         case 0: s=3 | 2<<2 | 1<<5; goto Join;
                                      case 1:
    when A;                           }
    pause;                                s=3 | 2<<3; goto Join;
    pause                         case 3: s=3 | 2<<4; goto Join;
                                  case 4:
when B                            }
                              s = 0; goto Join;
                              case 0: ; /* not running */
                              }
                              Join:
```

Figure 5

```
loop
    trap T in
        loop
            present A
            then
                emit B
            end;
            pause;
        end
    ||
        pause;
        exit T;
    end
end
```

```
if (inLaterCycles) {

    if (A) then B = 1;
    /* pause (level 1) */
||
    /* exit T (level 2) */

}

    if (A) then B = 1;
    /* pause (level 1) */
||
    /* pause (level 1) */

inLaterCycles = 1;
```

Figure 6A

```
/* THE THREE MAIN DATA TYPES OF ACCFG: CNODE, PROCESS, and THREAD.
MAIN DATA TYPE OF SCFG IS SNODE.

cnode = node in the acyclic concurrent control-flow graph (accfg)
snode = node in the sequential control-flow graph (scfg) */

/* The properties of a cnode are defined as follows: */

cnode::pthreads;    /* Threads to which this cnode belongs ("parents") Most
nodes belong to exactly one thread. The exceptions are join nodes, which belong to each
thread they join, and the topmost process, which belongs to no thread. */

/* A "predecessor" is a (snode, condition) pair that will be used as the source and label
respectively of an added arc. Each predecessor is an snode that could run a cnode */

cnode::runningPredecessors; /* set of normal snodes */
cnode::restartPredecessor;    /* restart snode */

/* The distinction between the two types of predecessor (i.e., "running" and "restart") is
used in the "suspend any running thread in process p" routine, which avoids creating
save state nodes for restart nodes. */

cnode::index;    /* integer index of the node. (topological order number) */

cnode::state;    /* Possible states are: Running, Runnable, or Suspended. Only a
"process" can be in a "Running" state, which means it contains a thread which is
actively executing. */
```

Figure 6B

/ A Process is a cnode (and therefore inherits the properties of a cnode) that corresponds to a fork node and contains one or more threads.*

A process's state may be Suspended, Runnable, or Running.

A Suspended process is contained in a thread that is not running.

A Runnable process is contained in a thread that is running, but none of the threads contained in the process are running. A Runnable process is ready to restart one of the threads it contains.

A Running process means one of its contained threads is currently running (i.e., executing instructions).

Suspending the running thread within a process changes the process's state from Running to Runnable. This is typically followed by starting (or restarting) another thread, contained within the process, which changes the process's state from Runnable back to Running. This suspension of one thread and the starting (or restarting) of another thread is also known as a "context switch."

*The properties of a process and a thread are as follows. */*

`process::threads;` */* The threads contained in the process */*

`process::runningThread;` */* Indicates which, if any, of the threads contained in the process is the currently running thread. */*

`thread::process;` */* Which process contains this thread */*

`thread::cnodes;` */* The cnodes in this thread that could be executed next */*

`thread::stateVariable;` */* State variable used for saving the state of the thread when the thread is suspended. This state variable is subsequently read when the thread is resumed. */*

Figure 6C

```
/* MAIN ROUTINE: "synthesize a scfg"
This main routine synthesizes the scfg from the input accfg */

synthesize a scfg
{
/* INITIALIZATION: Create the outermost process and a single thread within in. Put the
first scheduled node in this thread. The thread starts out suspended; the first iteration of
the main loop will resume it. */

en = create the SCFG entry node;

op = create the outermost process;

op.state = Runnable;

op.runningThread = none;

op.runningPredecessors += (en, -);   /* Entry node "en" is made to be
the runningPredecessor of "op" and the edge from op to en has no label as indicated by
the hyphen "-". */

op.pthreads = empty /* By definition, the outermost process is not in a thread. */

op.restartPredecessor = empty;

tt = new thread;

op.threads += tt;

tt.process = op;

fn = first node in the schedule;

/* Set the state variable used by the outermost thread */
tt.stateVariable = fn.index

tt.cnodes += fn;

fn.pthreads += tt; /* Put the first node in the top thread */

fn.state = Suspended;
```

Figure 6D

```
/* MAIN LOOP: successively assigns to current node "cn" each cnode of the input accfg
in order of the topological sort. */

for each node cn in scheduled order {

        sn = copy node cn and its expression into the SCFG;
        th = first thread in cn.pthreads; /* Thread of this node */

        /* Rest of this loop is divided into four main code blocks labeled A, B, C and D.
        For each cnode assigned to cn, a code block from A or B, and a code block
        selected from C or D, is executed.

        The pair of code blocks selected for execution depends on the type of the cnode,
        and is illustrated by the following table:

        cnode type:              Normal Fork Join
        selection from A or B:   B      B    A
        selection from C or D:   D      C    D

        if ( cn is a join node ) {
           /* CODE BLOCK A */
                /* Earlier, this join node would have been placed in all of the threads it
                was joining. Run it in its parent's thread. */
                p = th.process;
                th = thread in p.pthreads; /* unique since this is a process */
                switch to thread th;
                suspend any running thread in p;
                run cnode p as snode sn;
                th.cnodes -= p;    /* Delete the now-terminated process */

        } else { /* cn is a Normal or Fork node */
                /* CODE BLOCK B */
                switch to thread th;
                run cnode cn as snode sn;
                /* We've run cn, so it no longer plays a role in the thread */
                th.cnodes -= cn;
        }
```

Figure 6E

```
        if ( cn is a fork node ) {
            /* CODE BLOCK C */
            process = new process;
            process.state = Runnable;
            process.runningThread = none;
            process.runningPredecessors += (sn, -); /* Note that
            edge from "process" to sn has a empty label */
            process.restartPredecessor = empty;
            th.cnodes += process; /* Put the new process in its thread */
            for ( each successor cns of cn ) {
                /* Create a new thread for each successor and put the successor
                node in the new thread. */
                thread = new thread;
                process.threads += thread;
                thread.stateVariable = cns.index; /* Set the state
                variable for "thread" to have a default value being the topological
                index of cns. */
                thread.cnodes += cns;
                put cnode cns in thread thread;
                /* Initialize state of successor */
                cns.state = Suspended;
            }

        } else { /* This is a Normal or Join node */
            /* CODE BLOCK D */
            for ( each successor cns of cn ) {
                th.cnodes += cns;
                put cnode cns in thread th;
                cns.runningPredecessors += (sn, edge
                condition from cn to cns in the input accfg);
            } /* end "for ( each successor cns of cn )" */
        } /* end "else" */

    } /* end MAIN LOOP */

    } /* end "synthesize a scfg" */
```

Figure 6F

```
run cnode cn as snode sn
{
for ( each node snp in cn.runningPredecessors )
     add an edge from snp to sn, labeled like the
     predecessor edge from cn to snp;

if ( cn.restartPredecessor is not empty )
     add an edge from cn.restartPredecessor to sn, labeled
     like the predecessor edge from cn to
     cn.restartPredecessor;

/* having used these predecessor edges, they should now be removed */
cn.runningPredecessor = empty;
cn.restartPredecessor = empty;
}


put cnode cns in thread th
{
  if th is not already in cns.pthreads,
    cns.pthreads += th;
}
```

Figure 6G

```
switch to thread th
{
/* "switch to thread th" does nothing if the thread is already running. If the thread is not
running, it saves the state of any already-running thread (suspends it) and restarts the
desired thread. */

/* If there is at least one thread above "th," make sure it is also running */
if ( th.process.pthreads is not empty )
      /* The parent thread is unique for a process */
      switch to thread th.process.pthreads;

p = th.process;

/* If a different thread is running, suspend it */
if ( p.state == Running AND p.runningThread != th )
      suspend any running thread in p;

if ( p.state == Runnable ) {
      /* Restart our thread by adding a restart node and making this restart node a
      predecessor of each suspended node. */

      rn = new restart node( th.stateVariable );   /* Build a
      restart node (of SCFG) which tests state of the stateVariable for thread which is
      to be switched to. This stateVariable needs to have been set appropriately when
      thread th was previously suspended. */

      run cnode p as snode rn;

      for ( each cnode cn in th.cnodes ) {
            cn.restartPredecessor = (rn, cn.index); /* Create an
            edge from cn to rn whose label has the value cn.index */

            cn.state = Runnable;
      }

      p.state = Running;
      p.runningThread = th;

} /* end "if ( p.state == Runnable )" */

} /* end "switch to thread th" */
```

Figure 6H

```
suspend any running thread in process p
{
if ( p.state == Running ) {
        /* This process has a running thread -- suspend it */
        p.state = Runnable;
        th = p.runningThread;
        restartNode = none; /* Set when the restart node needs a default arc
        leading from it to suspend this thread */

        /* Save state if there is more than one running cnode in the thread */
        needToSaveState = true if there is more than one cnode
        in th;
        needToSaveState = false if there is not more than one
        cnode in th;

        /* Suspend each cnode in the the thread */

        for ( each cnode cn in th.cnodes ) {

                /* Suspend any running threads in a process node */
                if ( cn is a process )
                        suspend any running thread in cn;

                /* Suspend all running predecessors for this node */
                if ( cn.runningPredecessors is not empty ) {

                        if ( needToSaveState ) {
                                sn = new save state node (state for this
                                thread = cn.index ); /* Makes the "expression"
                                of sn be the following assignment statement:
                                th.stateVariable = cn.index. */

                                for ( each snode snp in
                                cn.runningPredecessors )
                                        add an edge from snp to sn, labeled
                                        like the predecessor edge from cn
                                        to snp;

                                cn.runningPredecessors = empty; /* having
                                used these predecessor edges, they should now be removed
                                */

                                p.runningPredecessors += (sn, -); /* add
                                an edge from p.runningPredecessors to sn, with no label */
```

Figure 6I

```
            } else { /* do not save state */
                for ( each snode snp in
                cn.runningPredecessors )
                    p.runningPredecessors += (snp, take
                    label from the edge cn to snp);

                cn.runningPredecessors = empty; /* having
                used these predecessor edges, they should now be removed
                */
            } /* end "else" */

        } /* end "if ( cn.runningPredecessors is not empty )" */

        /* Rembmer the restart node if this node has a restart predecessor. */

        if ( cn.restartPredecessor is not empty ) {
          restartNode = cn.restartPredecessor;
          /* Remove this precessor edge since it is empty */
          cn.restartPredecessor = empty;
        }

        cn.state = Suspended;

    } /* end "for ( each cnode cn in th.cnodes )" */

  p.runningThread = none;

  if ( restartNode is not none ) {
   /* At least one node had a restart predecessor: make sure an arc with a default
   condition is added from the restart node to handle this condition */
     p.runningPredecessors += (restartNode, -);
  }

} /* end if (p.state == Running) */

} /* end "suspend any running thread in process p" */
```

Figure 7

Input ACCFG

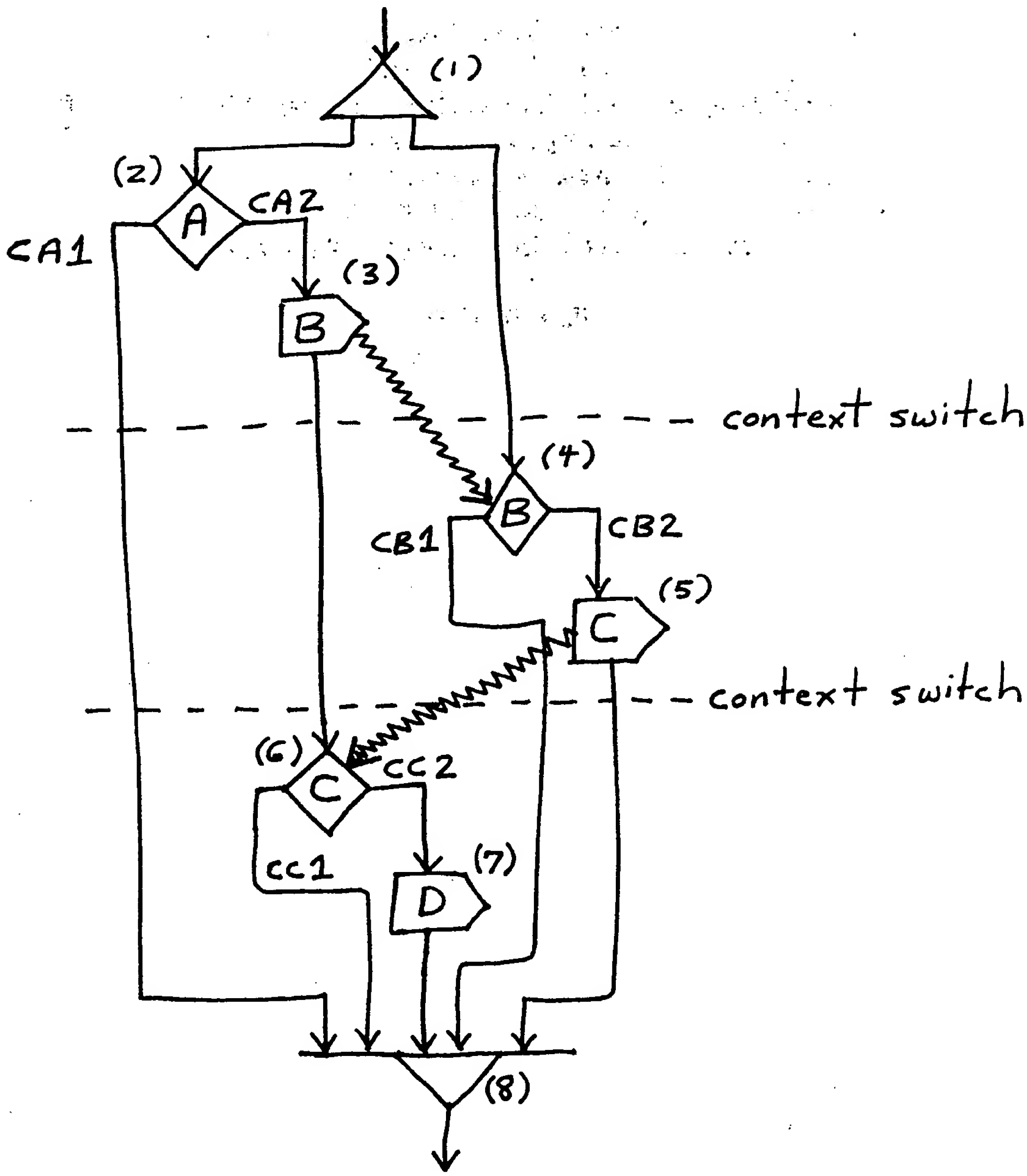


Figure 8A

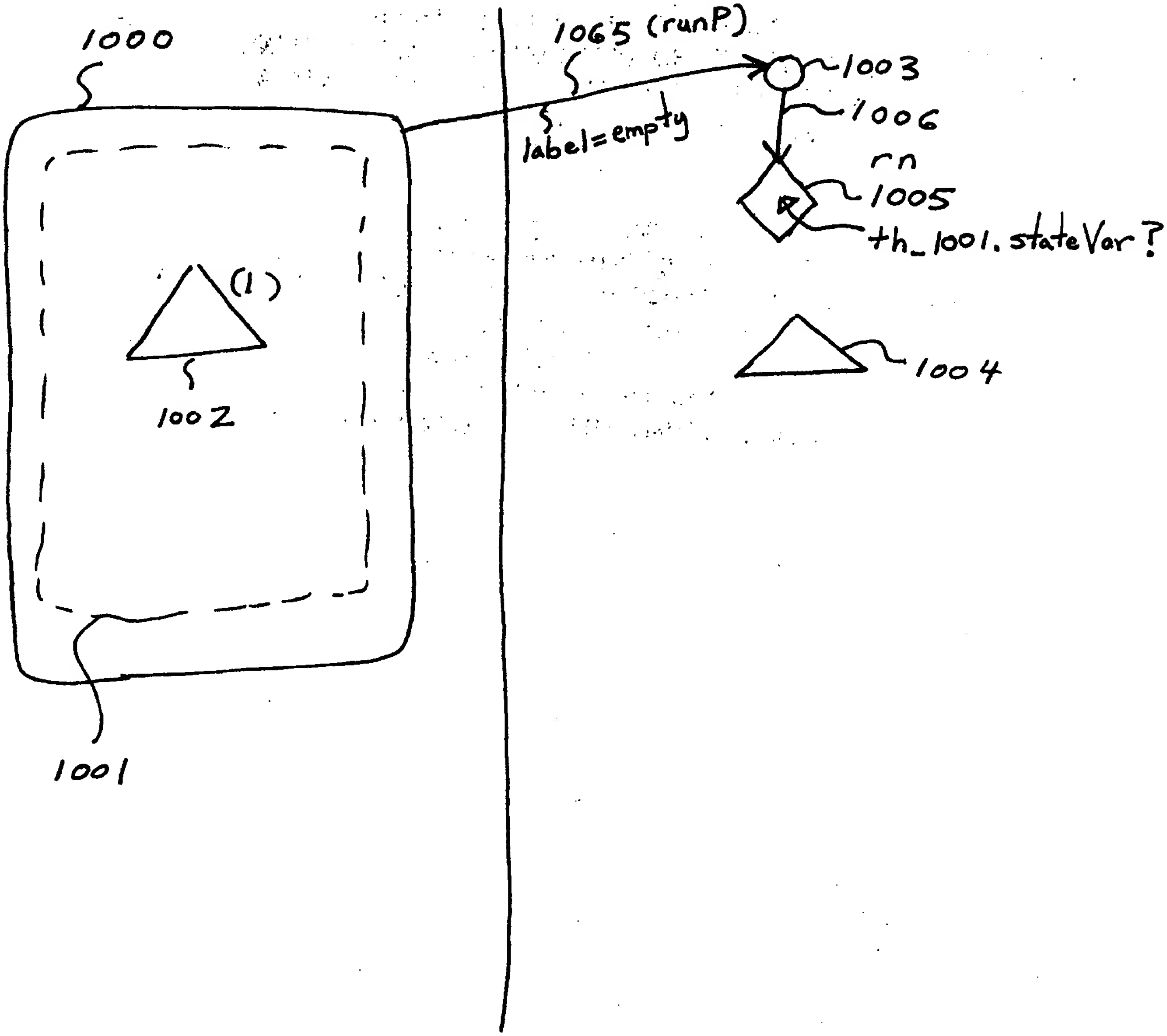

1000
1065 (runP)
1003
1006
label=empty
rn
1005
th_1001.stateVar?
(1)
1002
1004
1001

Figure 8B

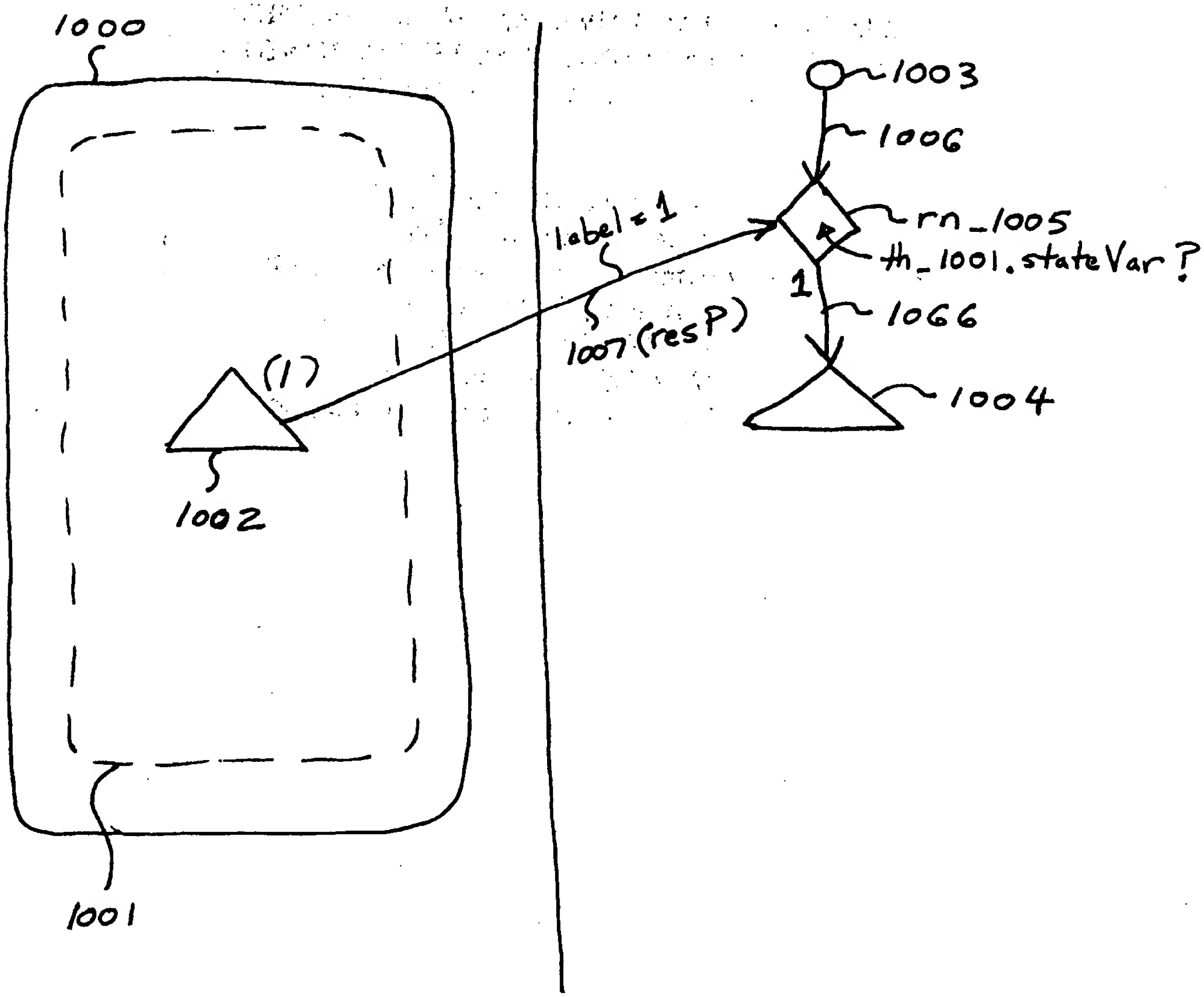

1000
1001
1002
(1)
label = 1
1007 (resP)
1003
1006
rn_1005
th_1001.stateVar?
1
1066
1004

Figure 8C

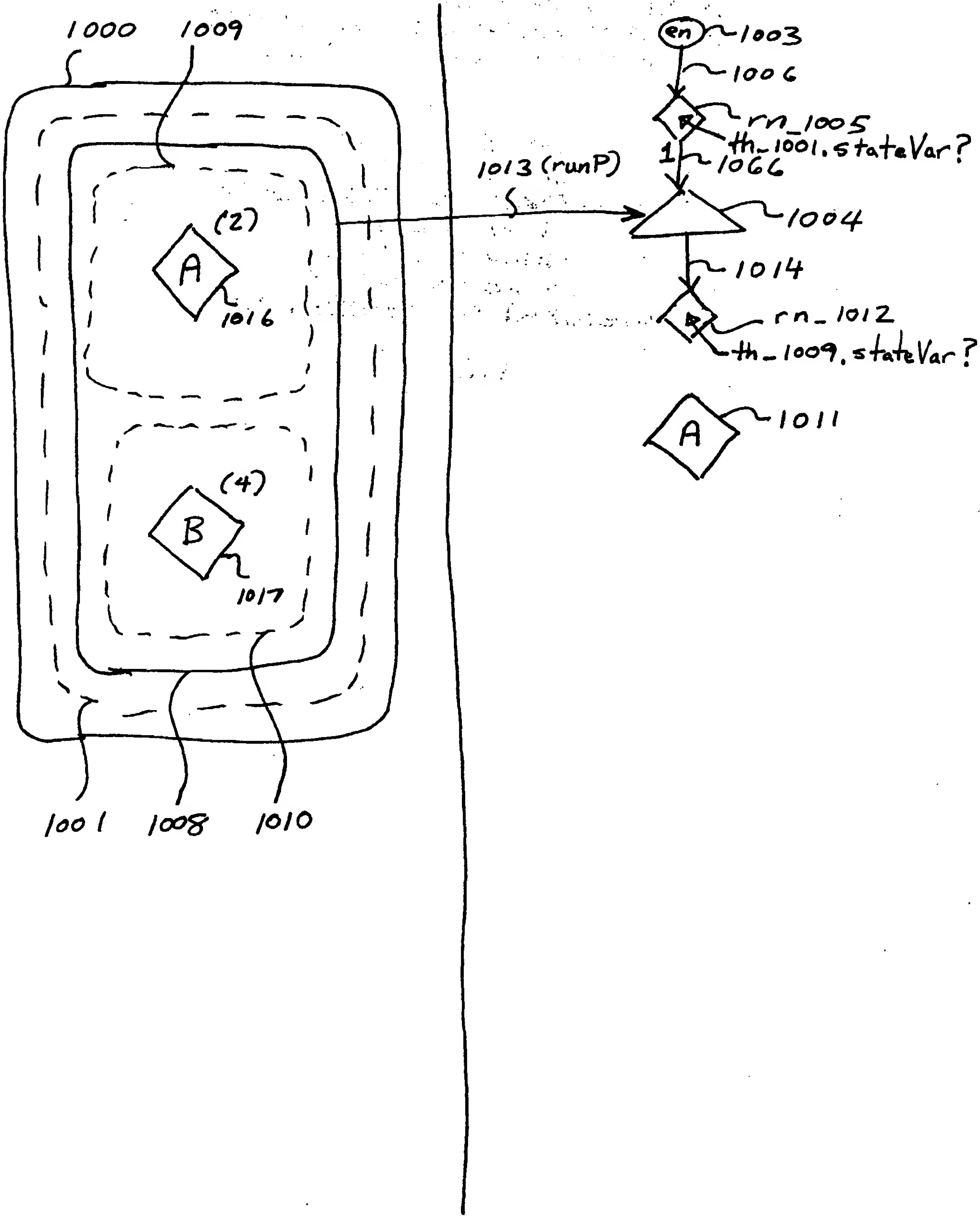
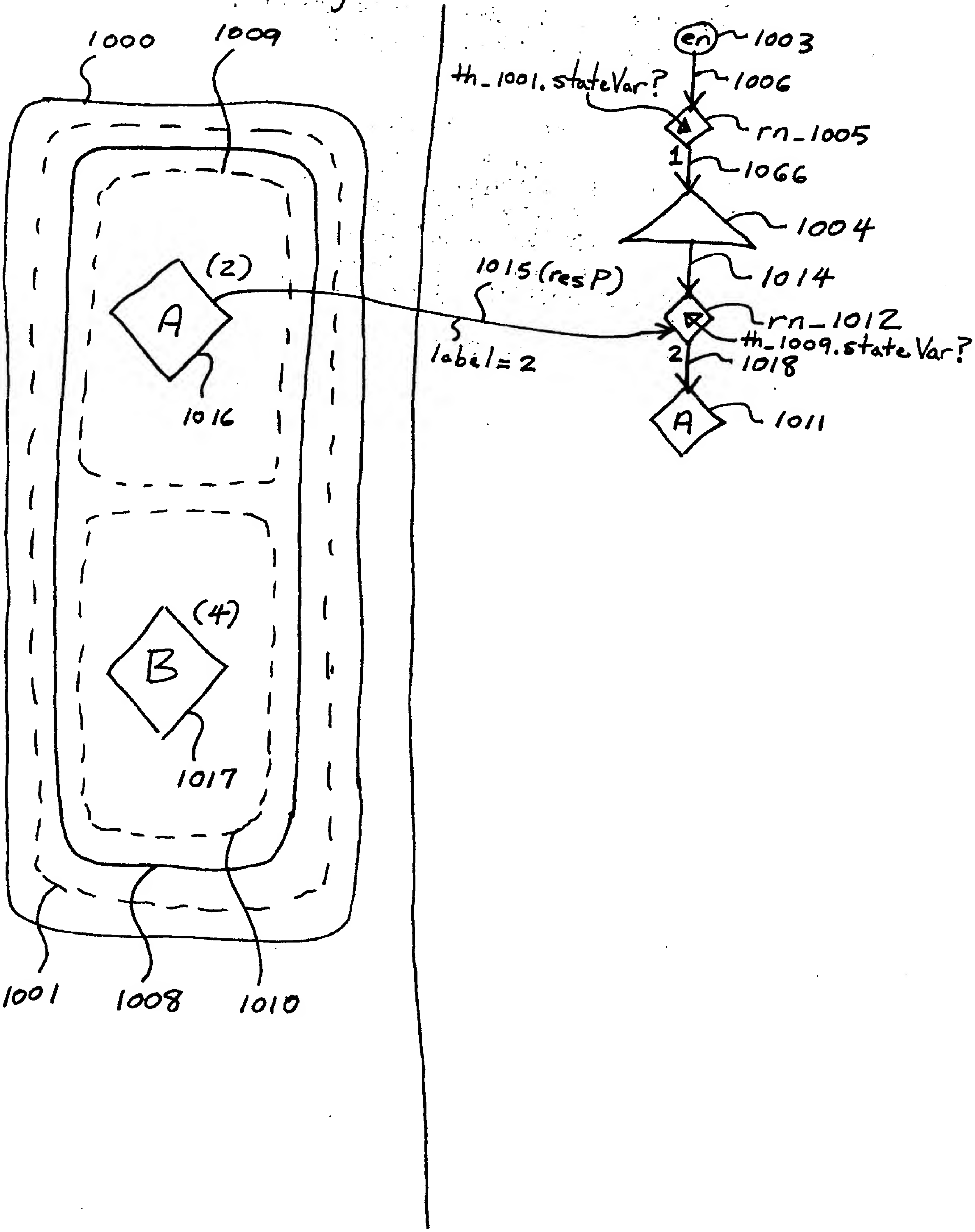
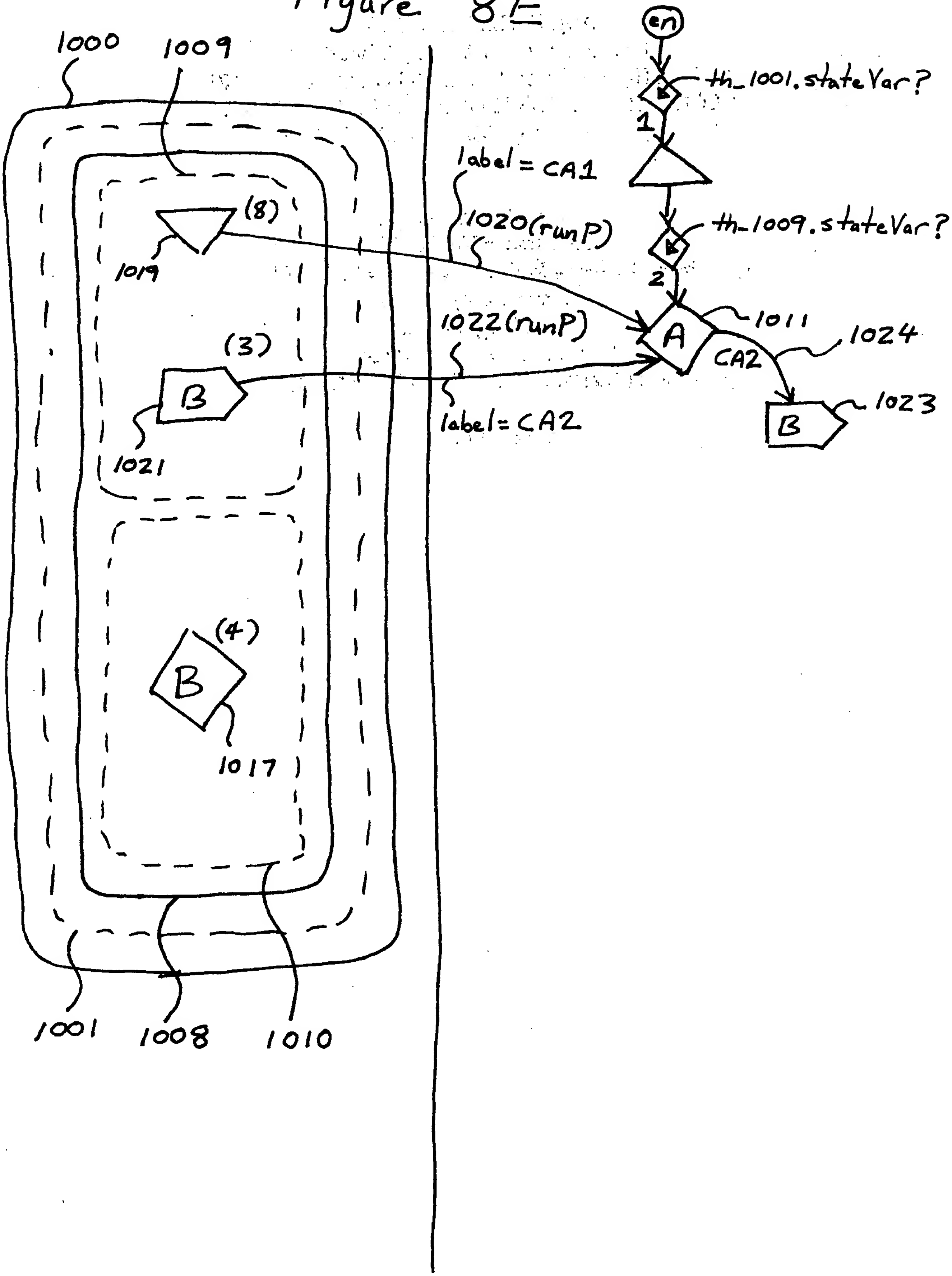

1000
1009
(2)
A
1016
(4)
B
1017
1001
1008
1010
1013 (runP)
en
1003
1006
rn_1005
th_1001.stateVar?
1
1066
1004
1014
rn_1012
th_1009.stateVar?
A
1011

Figure 8D



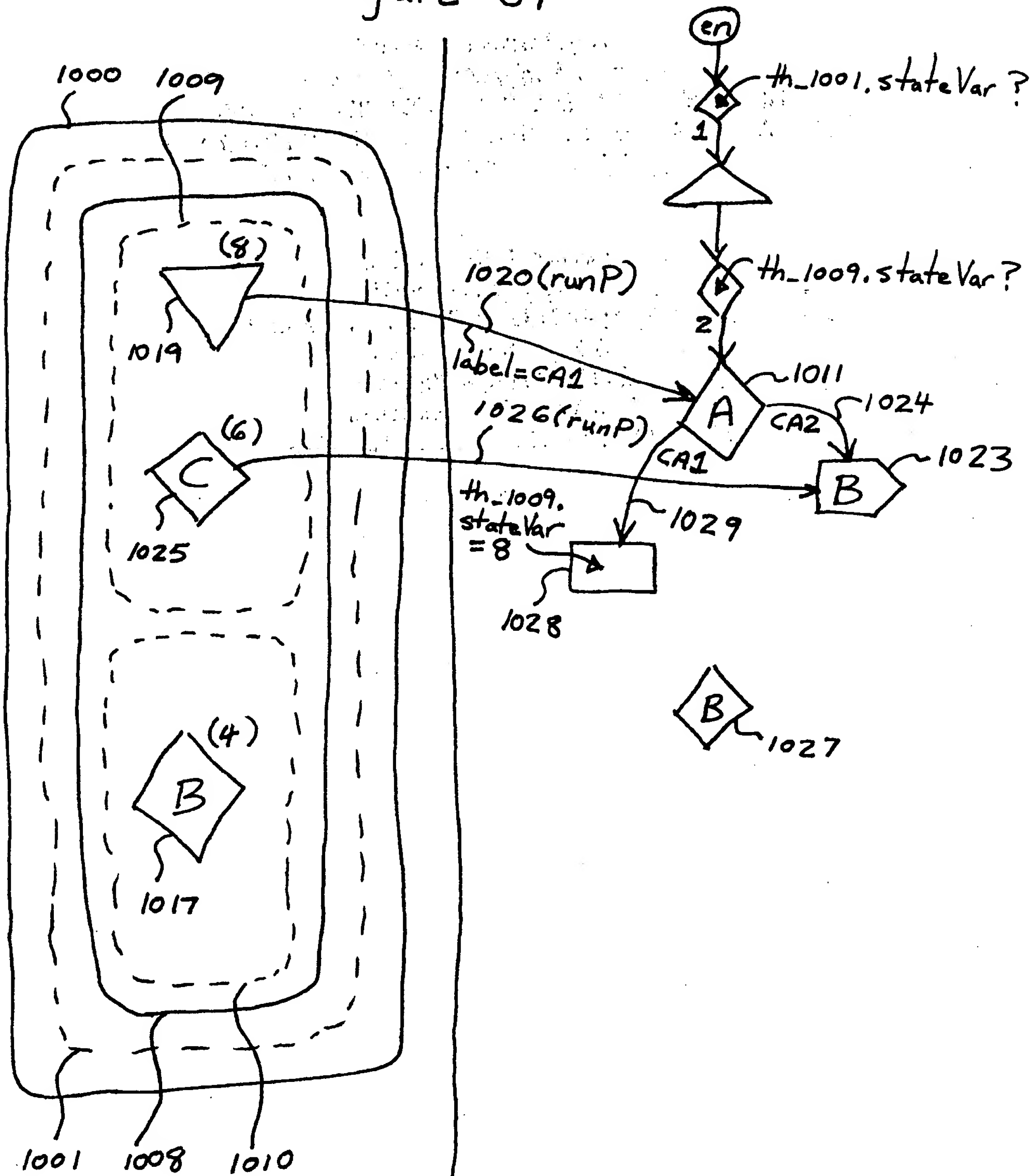
1000
1009
(2)
A
1016
(4)
B
1017
1001
1008
1010
1015 (resP)
label=2
en
1003
th_1001.stateVar?
1006
rn_1005
1
1066
1004
1014
rn_1012
th_1009.stateVar?
2
1018
A
1011

Figure 8E



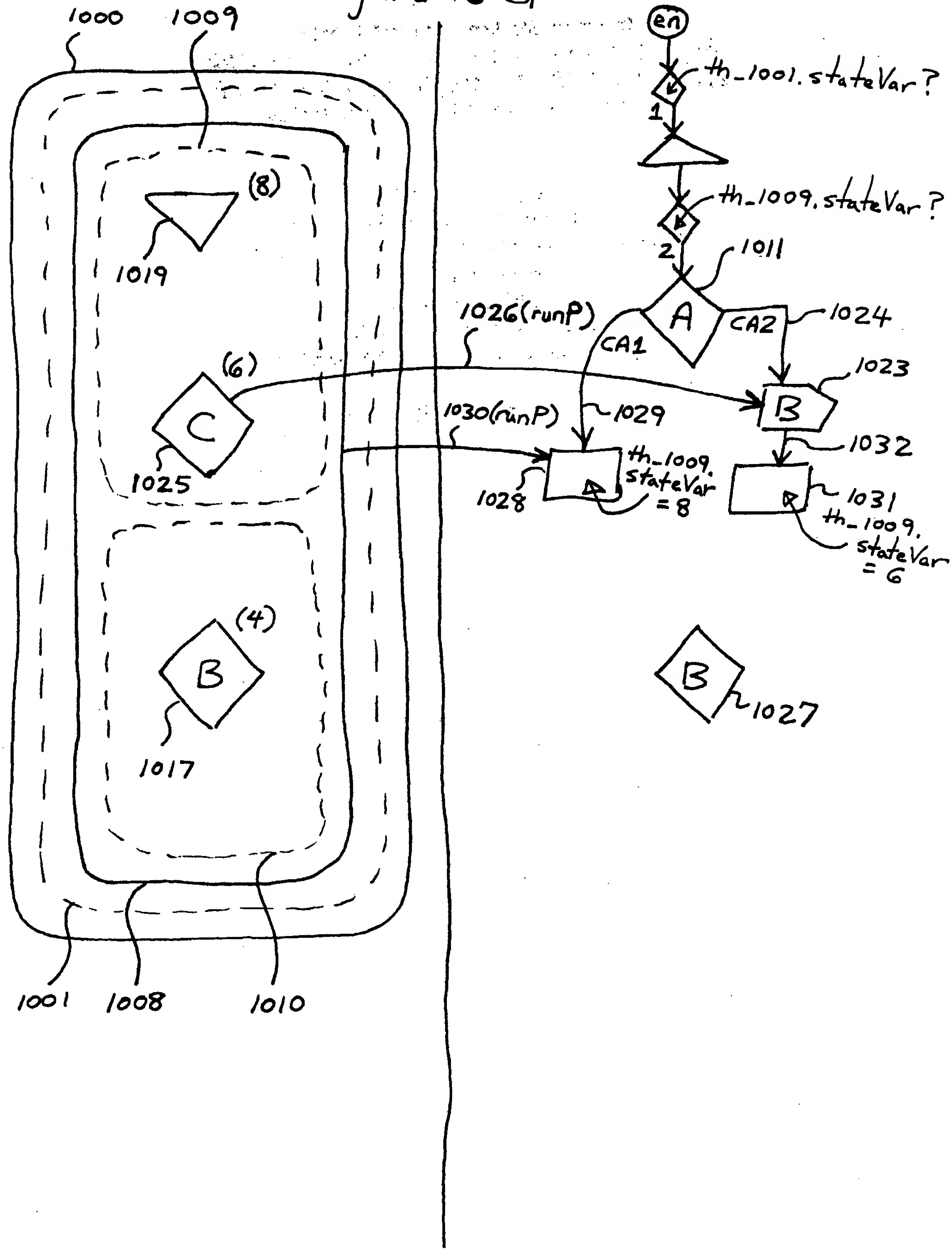
1000
1009
(8)
1019
(3)
B
1021
(4)
B
1017
1001
1008
1010
label = CA1
1020(runP)
1022(runP)
label = CA2
en
th_1001.stateVar?
1
th_1009.stateVar?
2
A
1011
CA2
1024
B
1023

Figure 8F



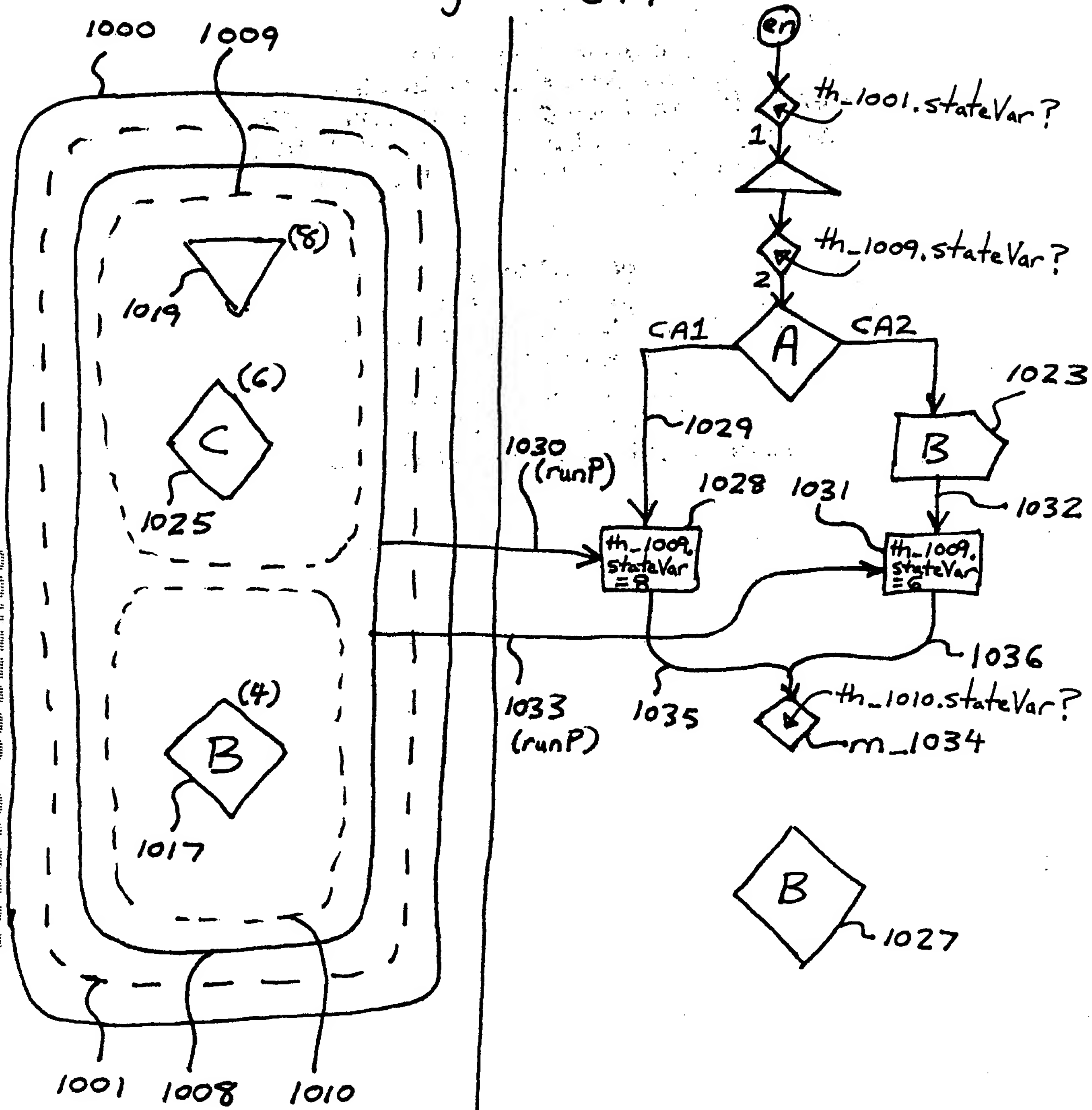
en
th_1001.stateVar?
1
th_1009.stateVar?
2
1000
1009
(8)
1019
1020(runP)
label=CA1
A
1011
CA2
1024
1026(runP)
CA1
(6)
C
1025
B
1023
th_1009.
stateVar
=8
1029
1028
B
1027
(4)
B
1017
1001
1008
1010

Figure 8G



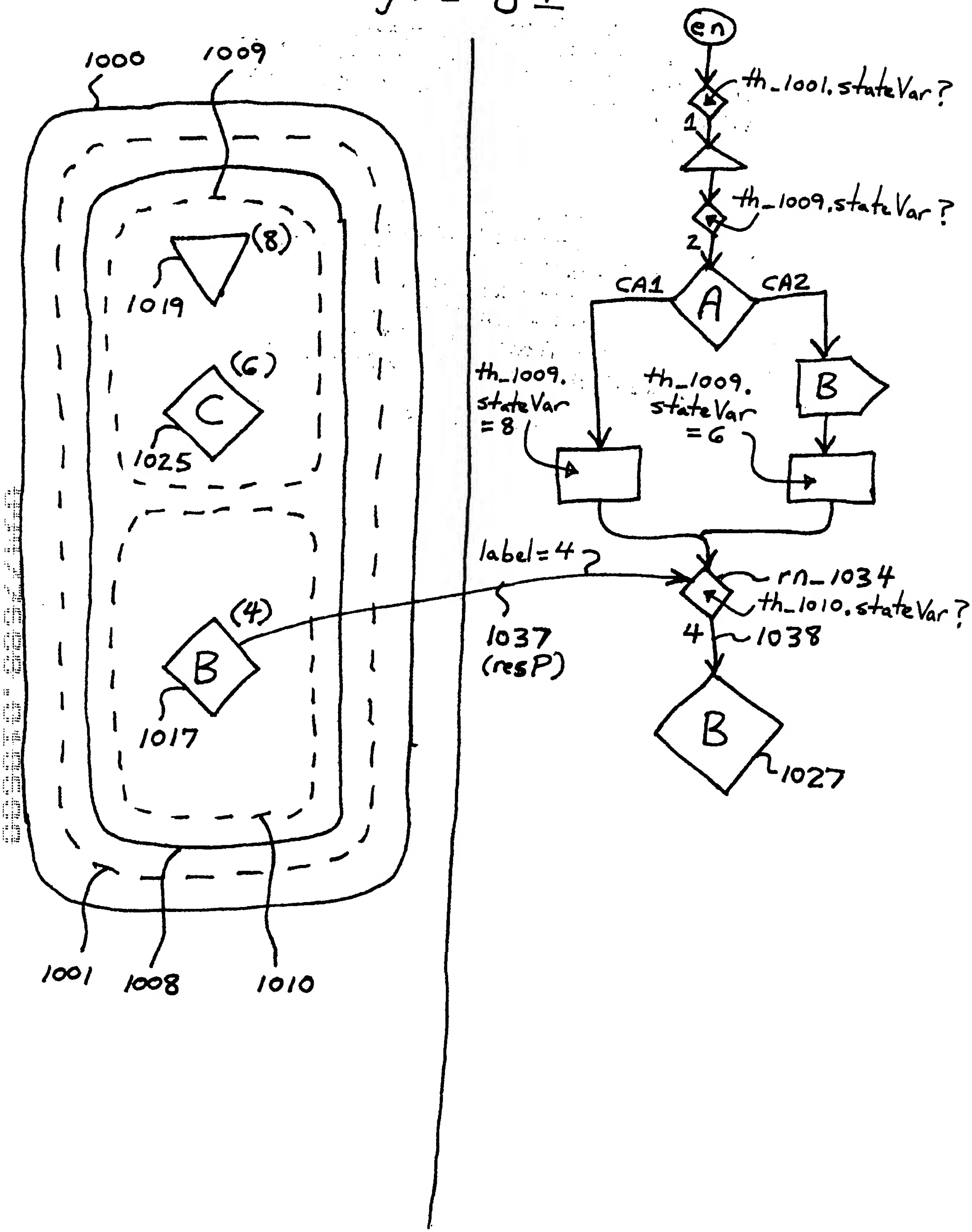
1000
1009
(8)
1019
(6)
C
1025
(4)
B
1017
1001
1008
1010
en
th_1001.stateVar?
1
th_1009.stateVar?
2
1011
A
CA1
CA2
1024
1026(runP)
1023
B
1030(runP)
1029
1032
th_1009.
stateVar
=8
1028
1031
th_1009.
stateVar
=6
B
1027

Figure 8H



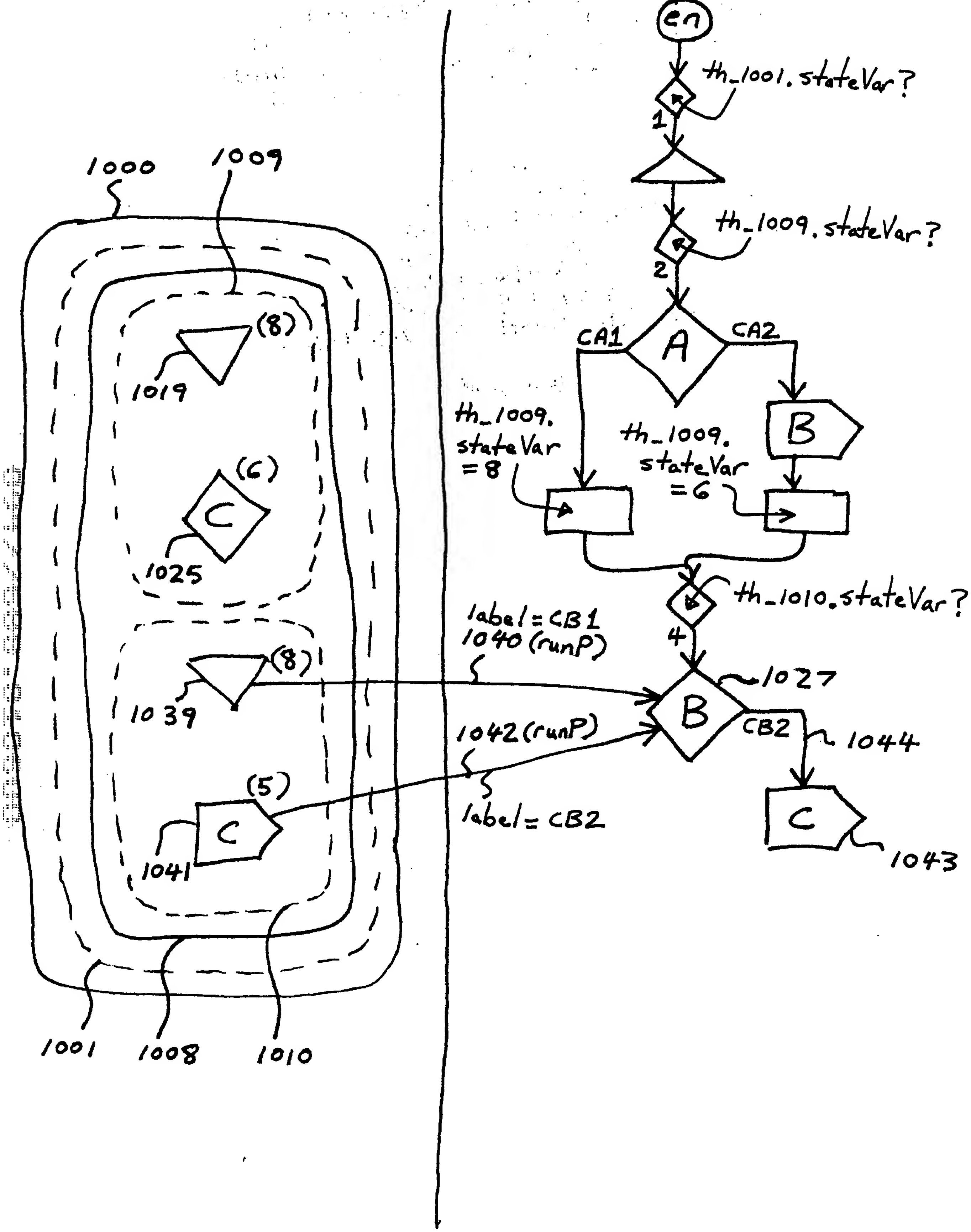
1000
1009
(8)
1019
(6)
C
1025
(4)
B
1017
1001
1008
1010
en
th_1001.stateVar?
1
th_1009.stateVar?
2
A
CA1
CA2
1023
B
1029
1030
(runP)
1028
1031
1032
th_1009.stateVar =8
th_1009.stateVar =6
1036
1033
(runP)
1035
th_1010.stateVar?
m_1034
B
1027

Figure 8I



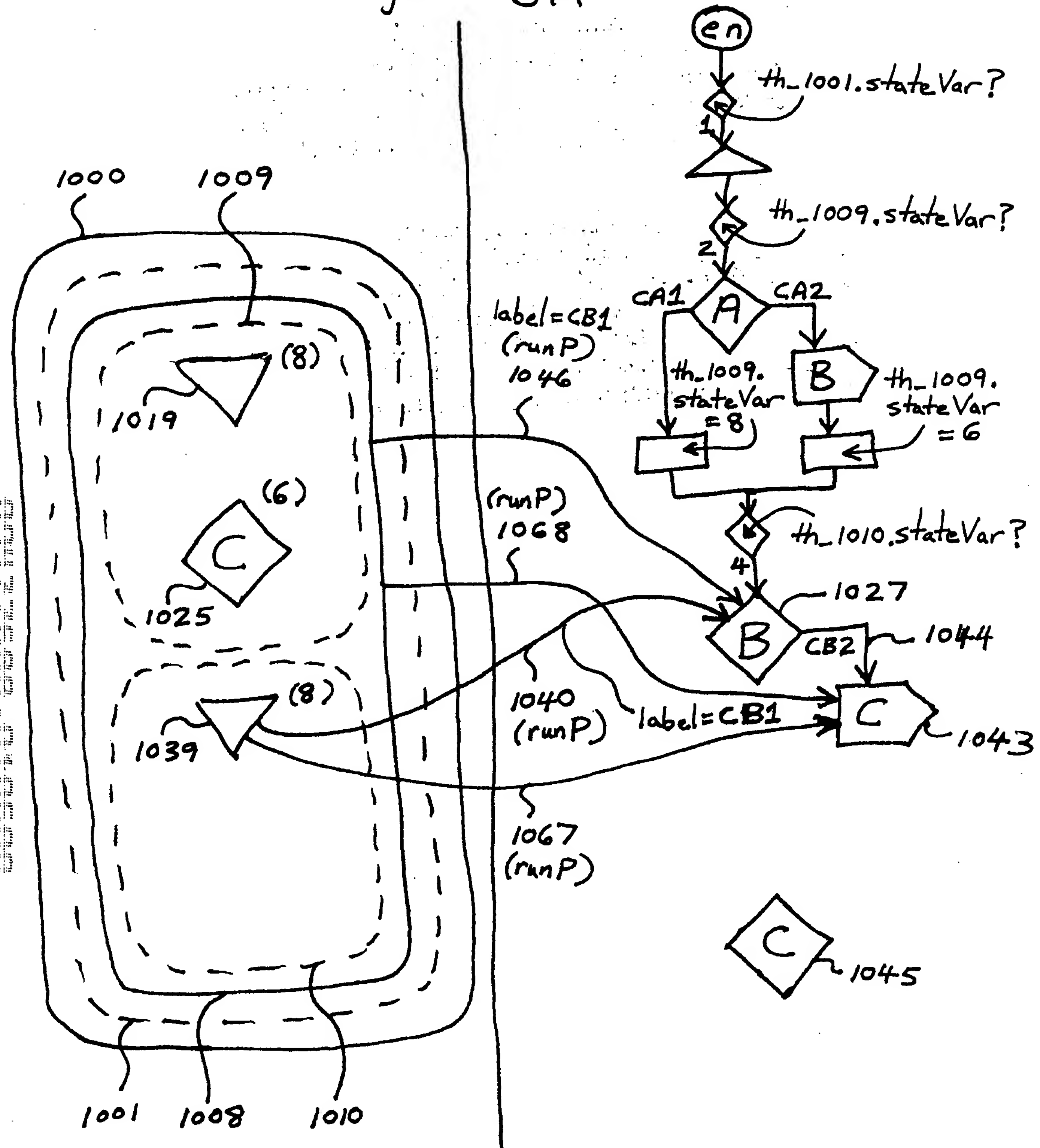
1000
1009
(8)
1019
(6)
C
1025
(4)
B
1017
1001
1008
1010
en
th_1001.stateVar?
1
th_1009.stateVar?
2
CA1
A
CA2
B
th_1009.
stateVar
=8
th_1009.
stateVar
=6
label=4
rn_1034
th_1010.stateVar?
1037
(resP)
4
1038
B
1027

Figure 8J



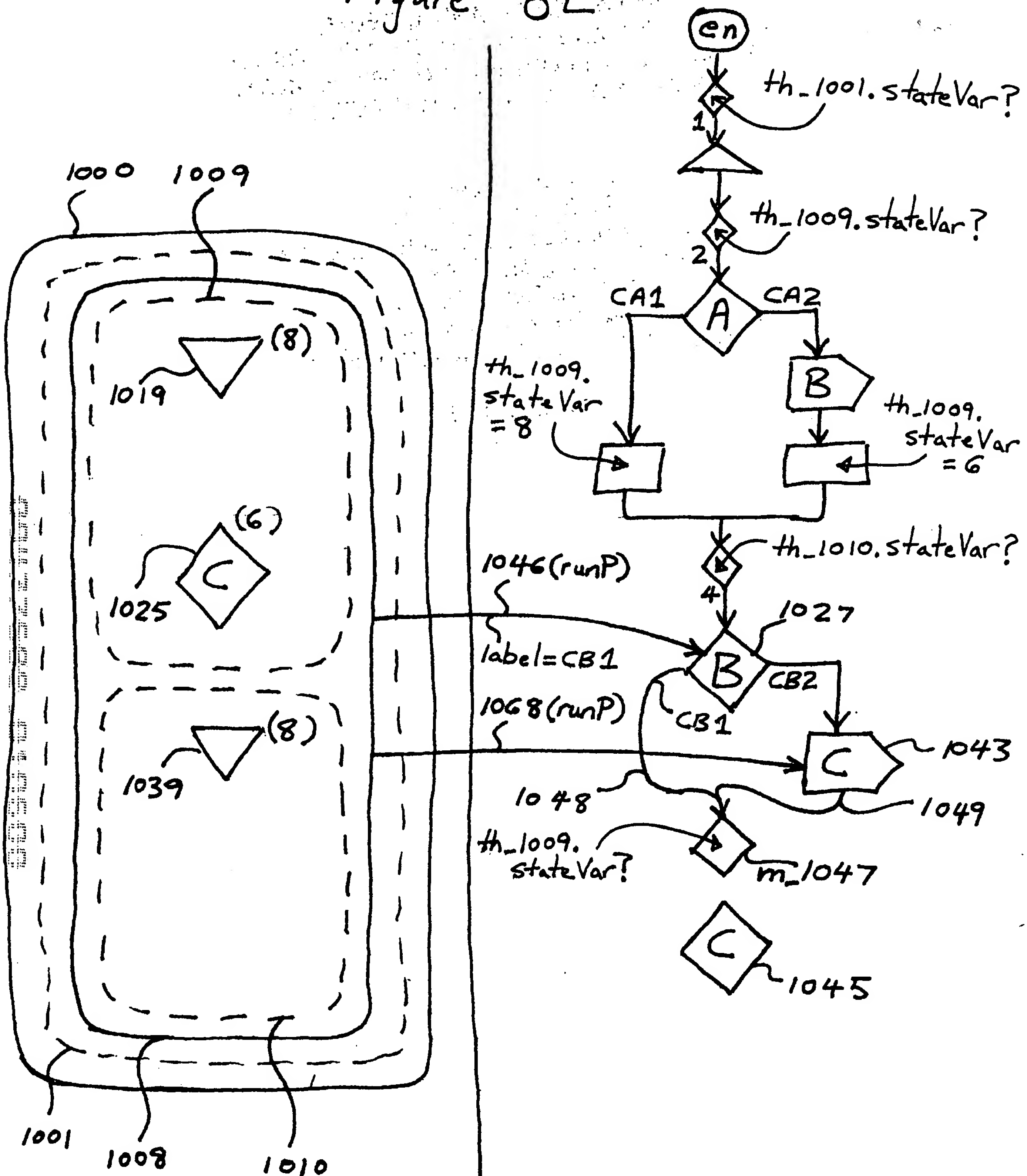
1000
1009
(8)
1019
(6)
C
1025
(8)
1039
(5)
C
1041
1001
1008
1010
en
th_1001.stateVar?
1
th_1009.stateVar?
2
CA1
A
CA2
B
th_1009.
stateVar
=8
th_1009.
stateVar
=6
th_1010.stateVar?
4
label=CB1
1040 (runP)
B
1027
1042(runP)
label=CB2
CB2
1044
C
1043

Figure 8K



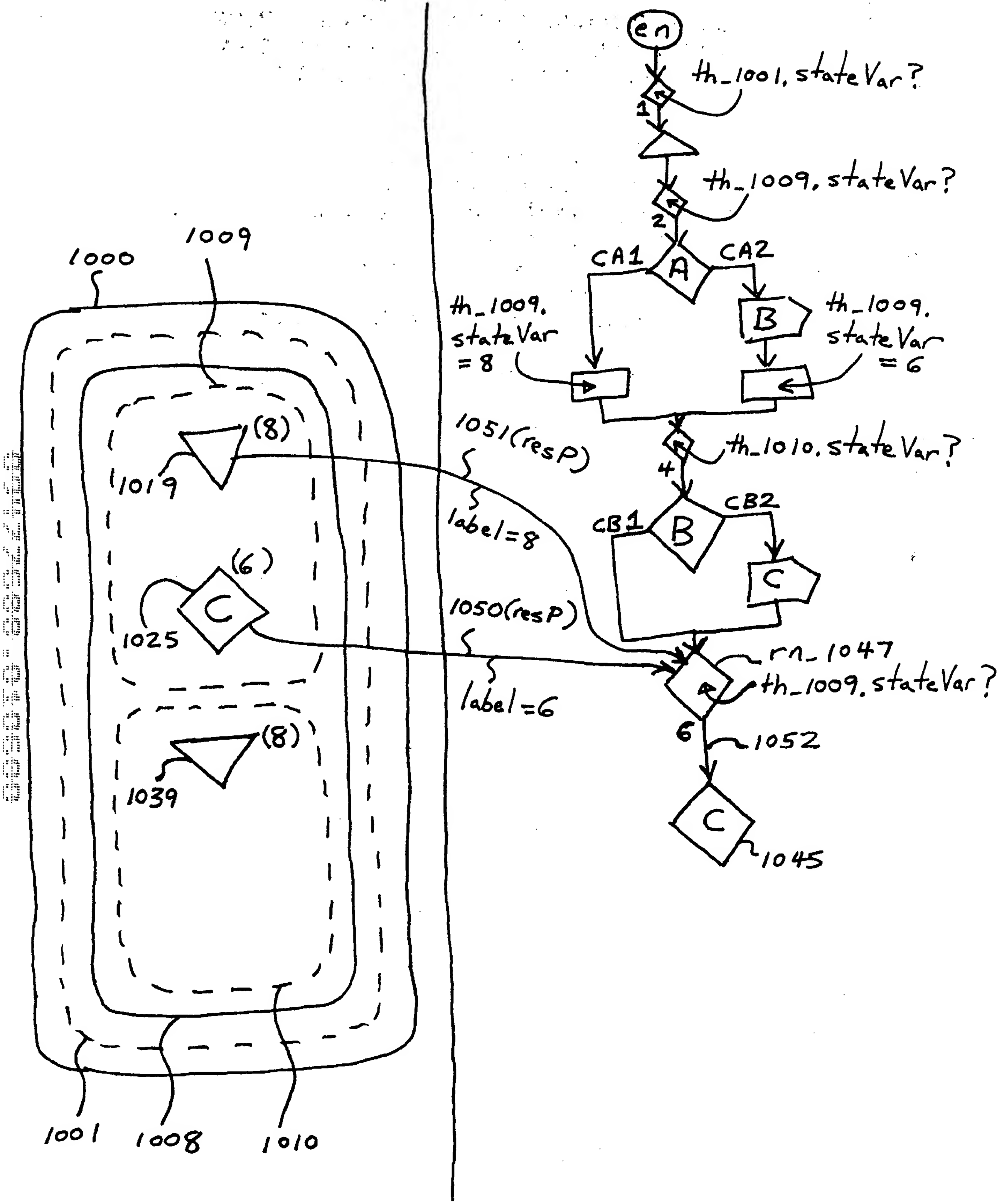
en
th_1001.stateVar?
1
th_1009.stateVar?
2
CA1
A
CA2
B
th_1009.
stateVar
=8
th_1009.
stateVar
=6
th_1010.stateVar?
4
1027
B
CB2
1044
C
1043
C
1045
1000
1009
(8)
1019
(6)
C
1025
(8)
1039
1001
1008
1010
label=CB1
(runP)
1046
(runP)
1068
1040
(runP)
label=CB1
1067
(runP)

Figure 8L



1000
1009
(8)
1019
(6)
1025
(8)
1039
1001
1008
1010
en
th_1001.stateVar?
1
th_1009.stateVar?
2
CA1
A
CA2
B
th_1009.
stateVar
=8
th_1009.
stateVar
=6
th_1010.stateVar?
4
1046(runP)
label=CB1
1027
B
CB2
CB1
1068(runP)
C
1043
1048
1049
th_1009.
stateVar?
m_1047
C
1045

Figure 8M



en
th_1001, stateVar?
1
th_1009, stateVar?
2
CA1
A
CA2
B
th_1009.
stateVar
= 8
th_1009.
stateVar
= 6
th_1010, stateVar?
4
CB1
B
CB2
C
rn_1047
th_1009, stateVar?
A
6
1052
C
1045
1051(resP)
label=8
1050(resP)
label=6
1000
1009
(8)
1019
(6)
C
1025
(8)
1039
1001
1008
1010

Fig 8N

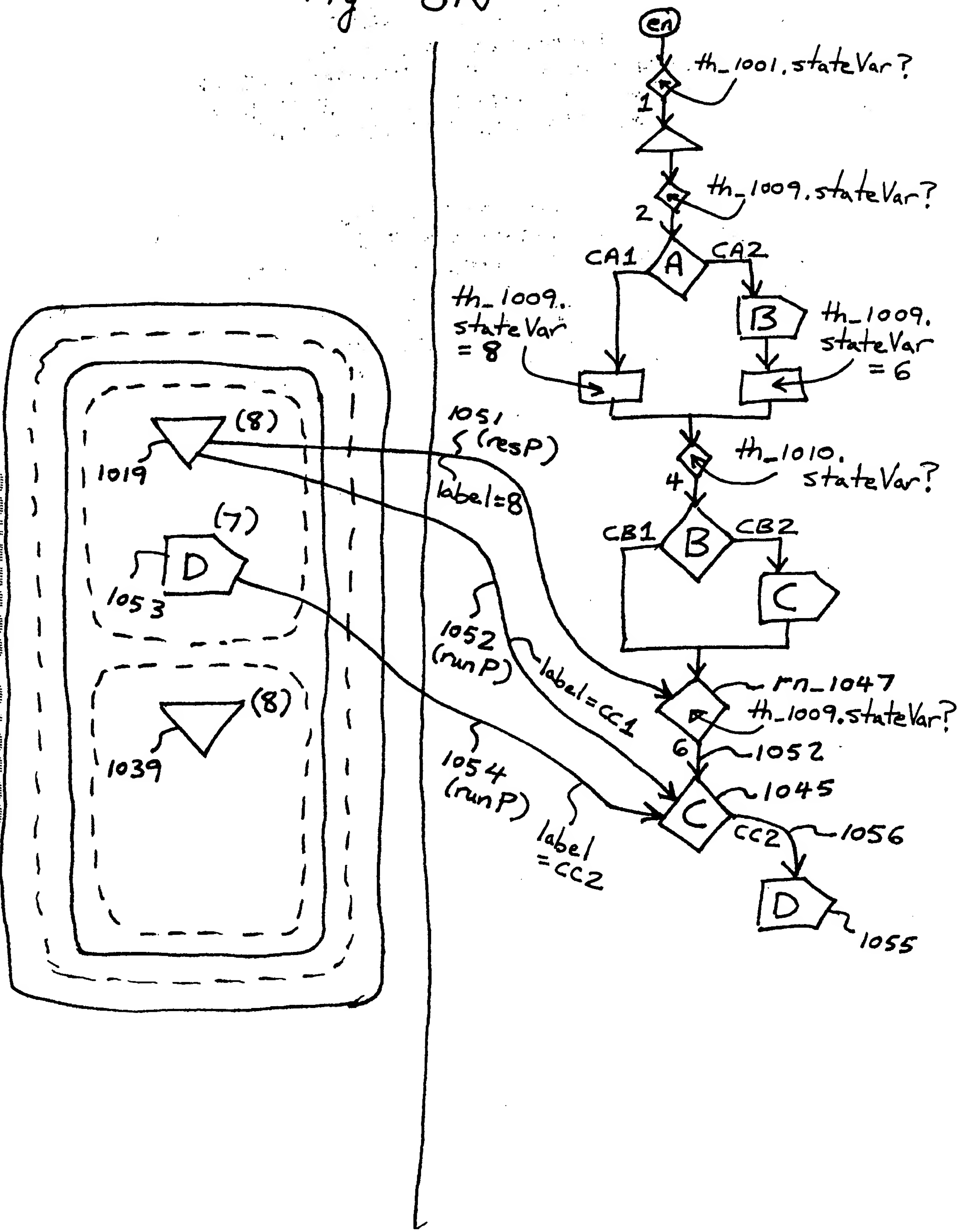

en
th_1001.stateVar?
1
th_1009.stateVar?
2
CA1
A
CA2
B
th_1009.
stateVar
= 8
th_1009.
stateVar
= 6
th_1010.
stateVar?
4
CB1
B
CB2
C
rn_1047
th_1009.stateVar?
6
1052
C
1045
CC2
1056
D
1055
1051
(resP)
label=8
1052
(runP)
label=CC1
1054
(runP)
label
=CC2
(8)
1019
(7)
D
1053
(8)
1039

Fig 8P

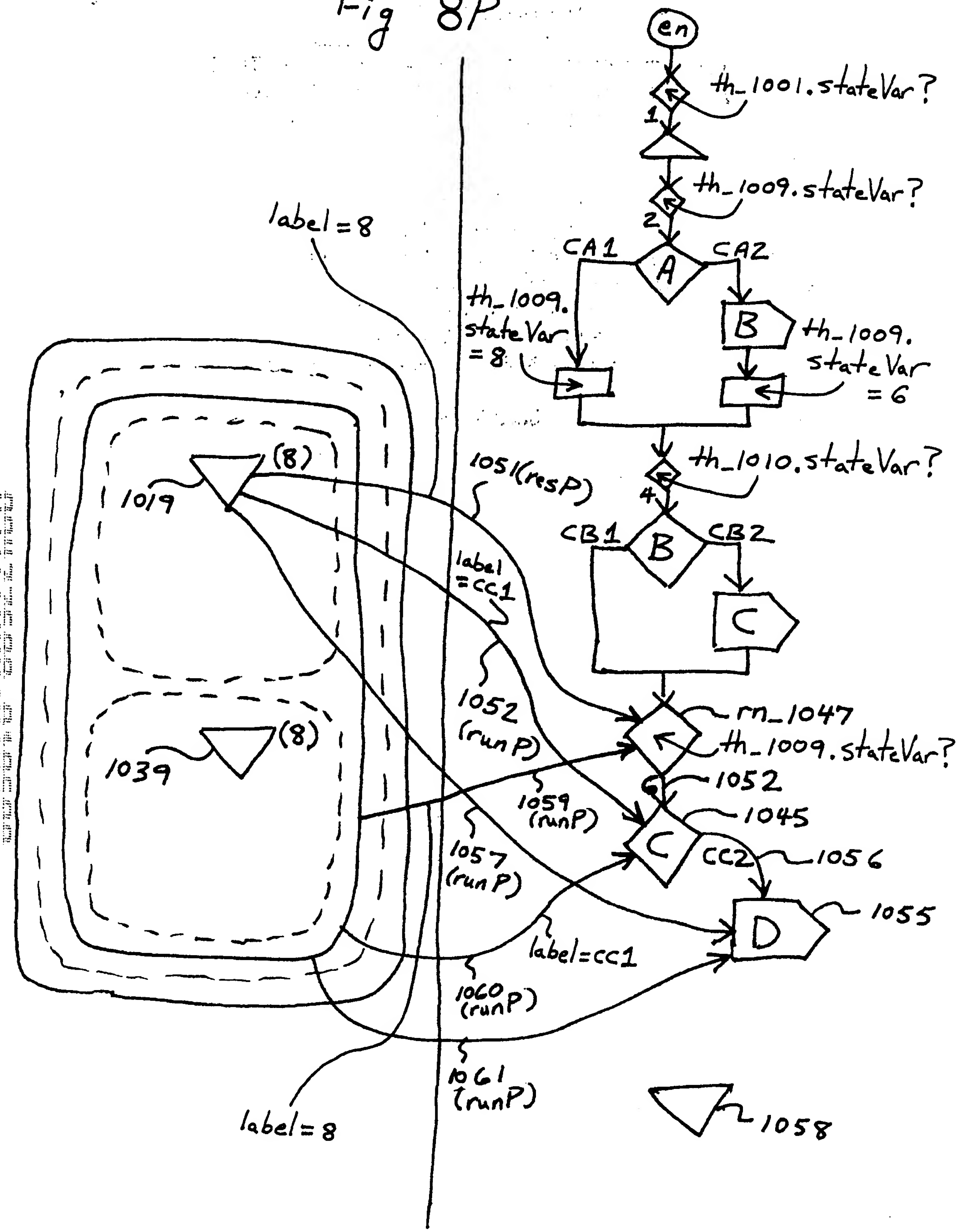
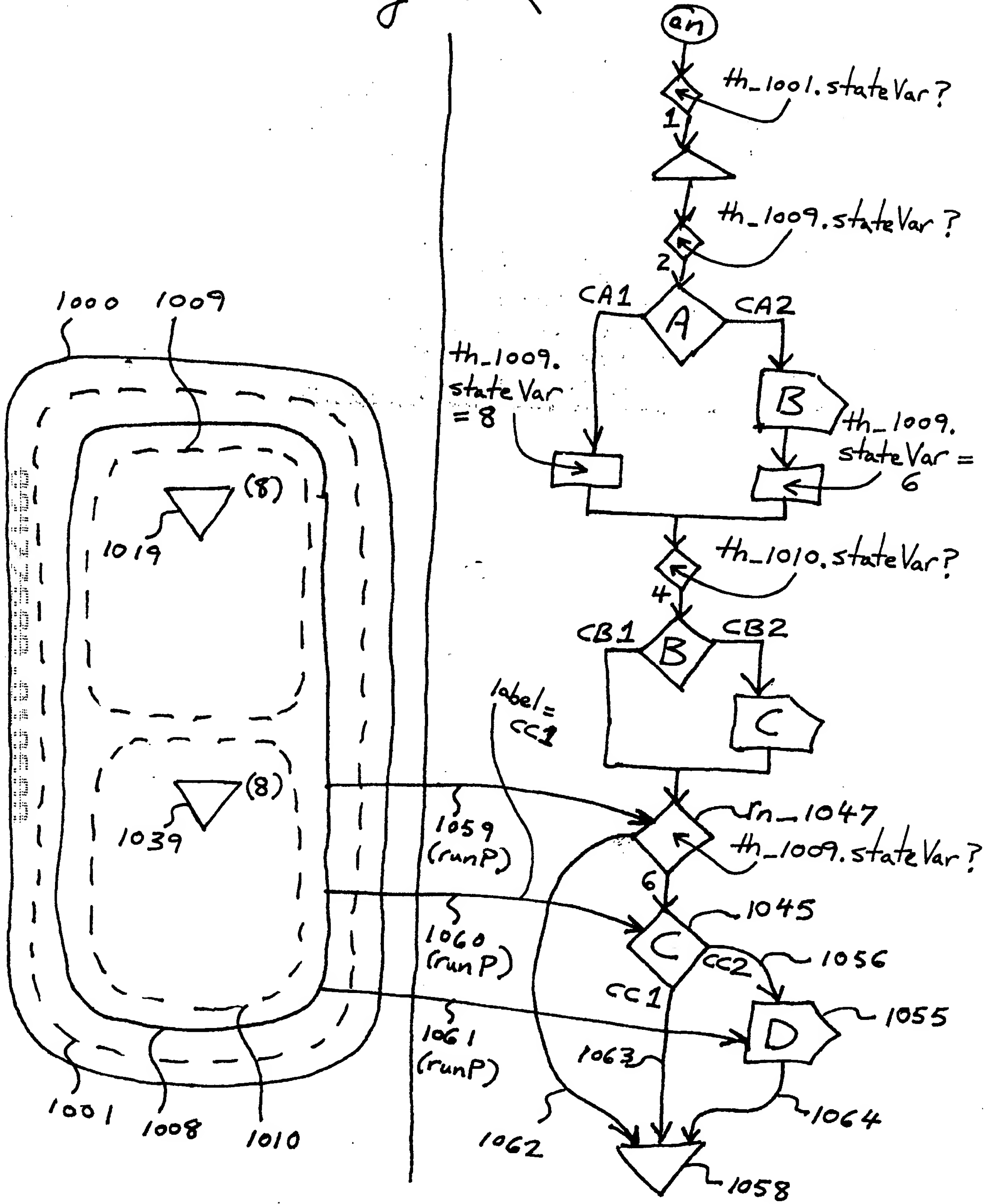

en
th_1001.stateVar?
1
th_1009.stateVar?
2
label=8
CA1
A
CA2
th_1009.
stateVar
=8
B
th_1009.
stateVar
=6
th_1010.stateVar?
4
1051(resP)
(8)
1019
label
=cc1
CB1
B
CB2
C
1052
(runP)
rn_1047
th_1009.stateVar?
(8)
1039
1052
1059
(runP)
1045
1057
(runP)
C
CC2
1056
D
1055
label=cc1
1060
(runP)
1061
(runP)
label=8
1058

Fig 8Q



an
th_1001.stateVar?
1
th_1009.stateVar?
2
CA1
A
CA2
B
th_1009.
stateVar
=8
th_1009.
stateVar =
6
th_1010.stateVar?
4
CB1
B
CB2
C
label=
cc1
rn_1047
th_1009.stateVar?
6
1045
C
CC2
1056
CC1
D
1055
1063
1064
1062
1058
1059
(runP)
1060
(runP)
1061
(runP)
1000
1009
(8)
1019
(8)
1039
1001
1008
1010

FIG. 9

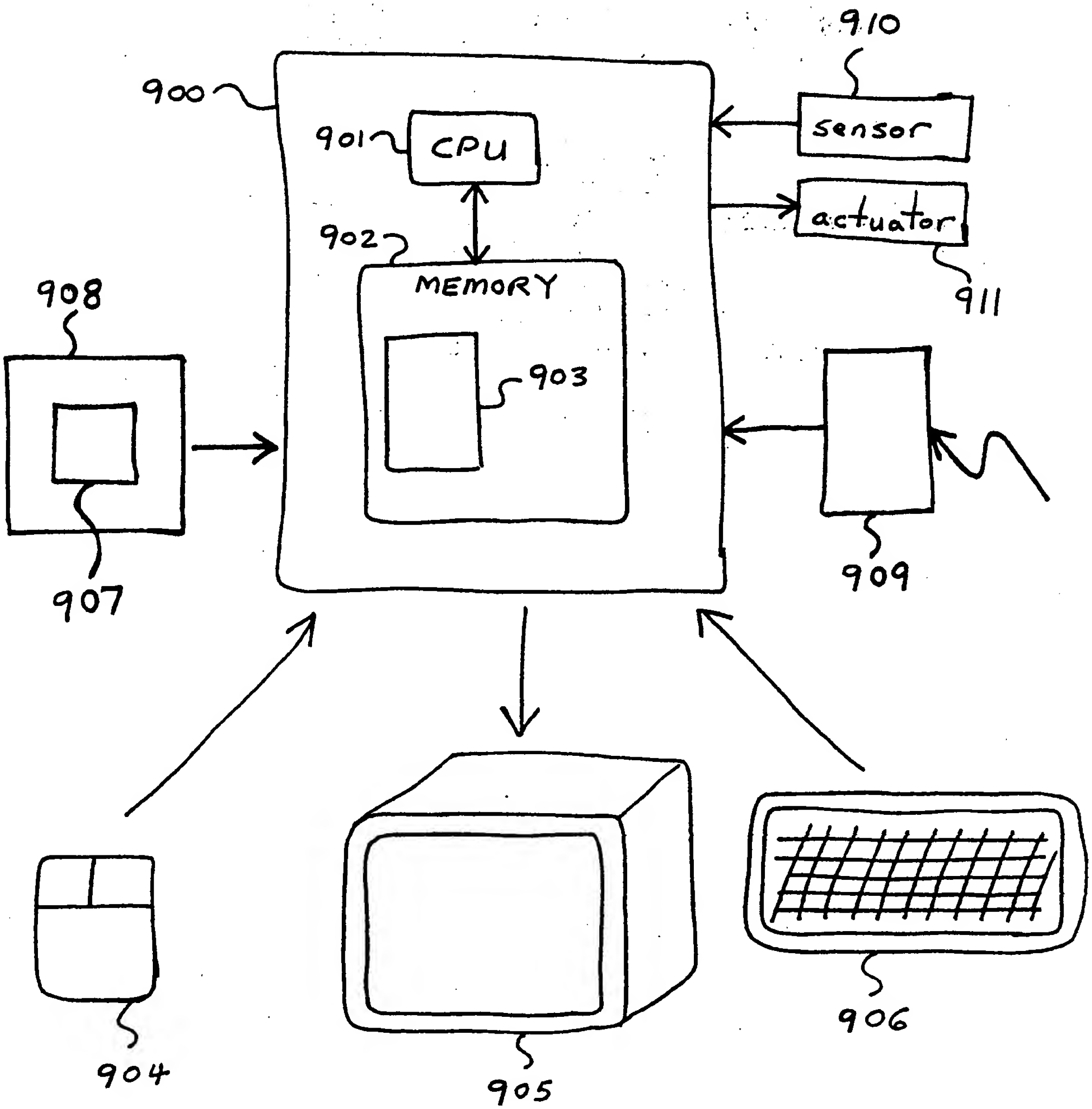
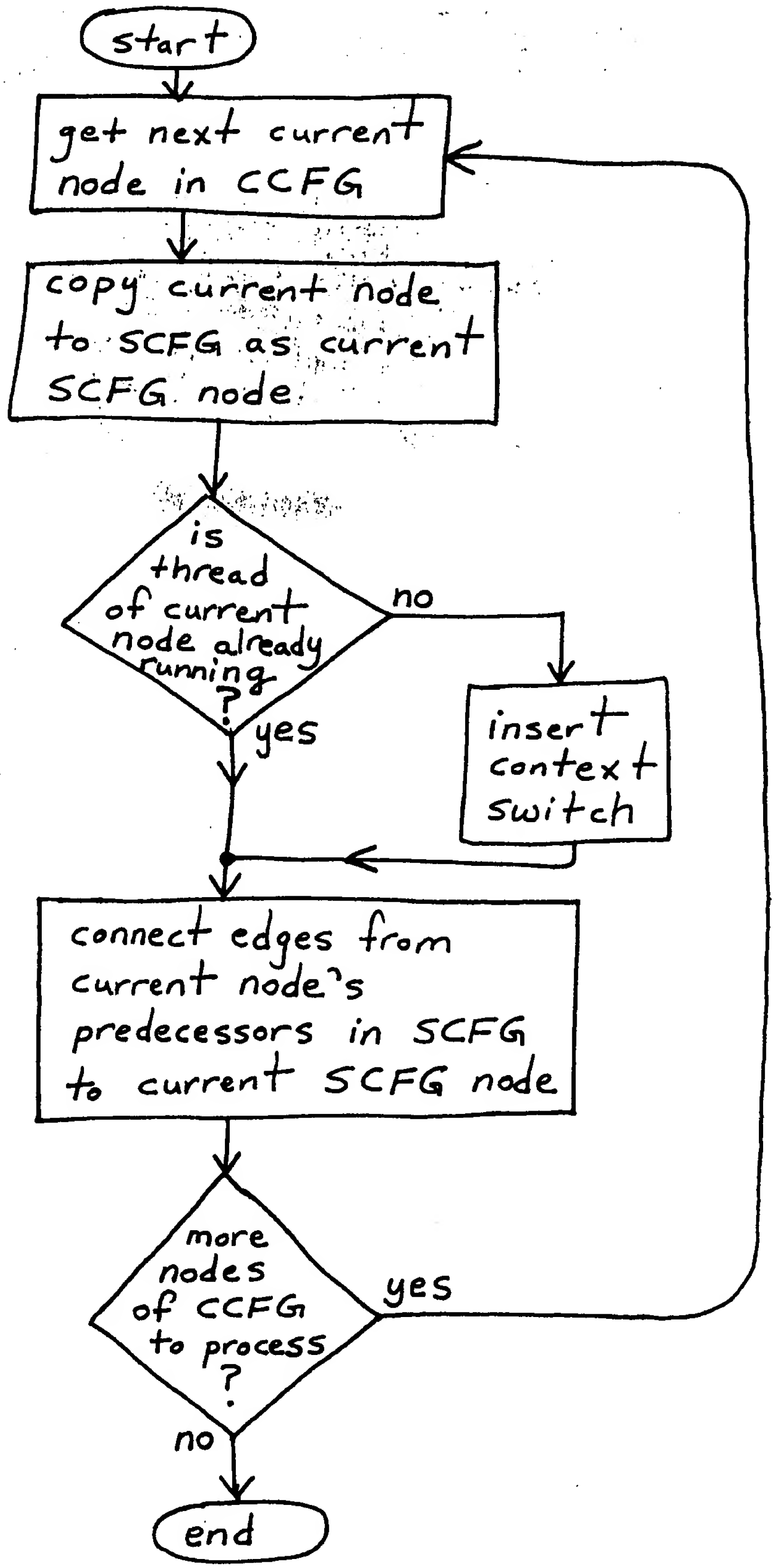

900
901
CPU
902
MEMORY
903
910
sensor
actuator
911
908
907
909
904
905
906

Figure 10


start
get next current node in CCFG
copy current node to SCFG as current SCFG node
is thread of current node already running ?
no
yes
insert context switch
connect edges from current node's predecessors in SCFG to current SCFG node
more nodes of CCFG to process ?
yes
no
end